Thread locals galore

The only thing more evil than one singleton is multiple singletons

Types of variables

- Locals (stored on the stack, or in registers)
- Heap allocations (stored on the heap)
- Statics (stored in a memory-mapped section of the executable!)
  - (process-local)
- Thread-local storage

How do thread-locals work?

Every "local" is at an offset in the "thread local storage" area

- x86_64-unknown-linux-gnu: **%fs** segment register
- aarch64-apple-darwin: **TPIDR_EL0**
- etc.

The location of the "thread local storage" area is updated whenever switching threads.

That's it!

Complication 1

Some types implement **Drop** — need to run them when the thread ends

Complication 2

Some types' initialization is not **const**: it needs to be evaluated at runtime

Complication matrix

```
//! Thread local support for platforms with native TLS.
//!
//! To achieve the best performance, we choose from four different types for
//! the TLS variable, depending on the method of initialization used (`const`
//! or lazy) and the drop requirements of the stored type:
//!
//! |         | `Drop`                | `!Drop`                |
//! |--------:|:----------------------:|:----------------------:|
//! | `const` | `EagerStorage<T>`     | `T`                    |
//! | lazy    | `LazyStorage<T, ()>` | `LazyStorage<T, !>` |
//!
```

Ultimately: one single interface

(Hopefully optimized!)

Doesn't matter how the underlying storage works.

```rust
#[cfg_attr(not(test), rustc_diagnostic_item = "LocalKey")]
#[stable(feature = "rust1", since = "1.0.0")]
pub struct LocalKey<T: 'static> {
    // This outer `LocalKey<T>` type is what's going to be stored in statics,
    // but actual data inside will sometimes be tagged with #[thread_local].
    // It's not valid for a true static to reference a #[thread_local] static,
    // so we get around that by exposing an accessor through a layer of function
    // indirection (this thunk).
    //
    // Note that the thunk is itself unsafe because the returned lifetime of the
    // slot where data lives, `'static`, is not actually valid. The lifetime
    // here is actually slightly shorter than the currently running thread!
    //
    // Although this is an extra layer of indirection, it should in theory be
    // trivially devirtualizable by LLVM because the value of `inner` never
    // changes and the constant should be readonly within a crate. This mainly
    // only runs into problems when TLS statics are exported across crates.
    inner: fn(Option<&mut Option<T>>) -> *const T,
}
```

Where are thread-locals used?

- Async runtimes
  - Need to register timers, interest in I/O events, etc.
- Anything with a "register"
  - tracing-subscriber

Sane case: single binary

Single crate, depends on tokio:

- One copy of tokio code baked into **binary**
- tokio's CONTEXT thread-local defined in **binary**

Sane case: crate-type = ["dylib"]

Some crate has type "dylib" and depends on tokio

- One copy of tokio code baked into **libtokio.so**
- tokio's CONTEXT thread-local defined in **libtokio.so**
- **binary** (and any other **.so** files) all depend on **libtokio.so**

My case: crate-type = ["cdylib"]

One **binary** crate, and several "modules", built *separately* as **libmodfoo.so** and **libmodbar.so** — loaded dynamically at startup.

- N copies of tokio code, one in each object:
  - once in **binary,** once in **libmodfoo.so,** once in **libmodbar.so**
- N copies of tokio's CONTEXT thread-local:
  - once in **binary**, once in **libmodfoo.so**, once in **libmodbar.so**

Solution: pass tokio context explicitly

Problem: not all crates accept an explicit "Executor"!

Can we just set the module's copy of tokio's CONTEXT thread-local?

Yes! Wrapper around Future that enters a runtime Handle every time.

But: **spawning task that spawns a task defeats that.**

Solution: patch tokio

- Define tokio's CONTEXT thread-local **only once**
  - In **binary**
- Enable **external-tls** feature in tokio for modules
  - ...which redefines the **tokio_thread_local!** Macro

Solution: patch tokio

- Define **TOKIO_CONTEXT** in **tls-slots** crate (of type **dylib**)

```
lith on  iggy-stuff [+]
❯ nm modtest/target/debug/libtls_slots.dylib | grep "D _CONTEXT"
000000000023c898 D _CONTEXT_THUNK
```

- Have **binary** depend on **tls-slots**

```
lith on  iggy-stuff [+]
❯ nm modtest/target/debug/modtest | grep "_CONTEXT"
                 U _CONTEXT_THUNK
```

- Allow mods to have undefined symbols with **-undefined dynamic_lookup**

Does it work?

Almost!

Statics ("process-local" variables) remain, and they seem to be messing with the multi-threaded executor.

(Workaround: spawn a future looping for 10ms in a loop — still doesn't work all the time)

The current_thread executor is happy though!

Next steps

- Clean up solution (remove some indirection?)
- Apply same treatment to statics / "process-locals"
- Tackle **tracing-subscriber**

Will this get upstreamed? Unlikely. This is a cursed scenario. But who knows.